Efficient Higher-Order Derivatives of the Hypergeometric Function

I. Charpentier[1], C. Dal Cappello[2], and J. Utke[3]

[1] Laboratoire de Physique et Mécanique des Matériaux, UMR CNRS 7554, Ile du Saulcy, 57045 Metz Cedex 1, France; isabelle.charpentier@univ-metz.fr
[2] Laboratoire de Physique Moléculaire et des Collisions, 1, Bd Arago, 57078 Metz Cedex 3, France; cappello@univ-metz.fr
[3] Argonne National Laboratory, Mathematics and Computer Science Division, 9700 South Cass Avenue, Argonne, Illinois 60439,USA; utke@mcs.anl.gov

Summary. Various physics applications involve the computation of the standard hypergeometric function ${}_2F_1$ and its derivatives. Because it is not an intrinsic in the common programming languages, automatic differentiation tools will either differentiate through the code that computes ${}_2F_1$ if that code is available or require the user to provide hand-written derivative code. We present options for the derivative computation in the context of an ionization problems and compare the approach implemented in the Diamant library to standard methods.

Key words: higher-order derivatives, hypergeometric function, Bell polynomials, Diamant library

1 Introduction

The Gauss hypergeometric function ${}_2F_1(a,b,c;z)$ frequently arises in physics problems such as ionization; see Sec. 3. It is a solution of the hypergeometric differential equation

$$z(1-z)\frac{d^2\varphi(z)}{dz^2}+[c-(a+b+1)z]\frac{d\varphi(z)}{dz}-ab\varphi(z)=0 \tag{1}$$

and, by using the rising factorial $(x)_n$, can be expressed as a series

$${}_2F_1(a,b,c;z)=\sum_{n=0}^{\infty}\frac{(a)_n(b)_n}{(c)_n}\frac{z^n}{n!} \quad \text{with} \quad (x)_n=\frac{(x+n-1)!}{(x-1)!} \tag{2}$$

The convergence of the series depends on (a,b,c) and z and is discussed in detail in [1]. For $k=1,\cdots,n$, differentiating (2) with respect to z yields

$${}_2F_1{}^{(k)}(a,b,c;z)=\frac{\partial^k {}_2F_1(a,b,c;z)}{\partial z^k}=\frac{(a)_k(b)_k}{(c)_k}{}_2F_1(a+k,b+k,c+k;z) \tag{3}$$

The ensuing n computationally expensive evaluations of ${}_2F_1$ at different arguments may be avoided by deriving a recurrence formula for the Taylor coefficients from (1). In automatic

differentiation (AD) this approach has been the basis for computing higher-order Taylor coefficients for intrinsic functions such as e^x; see [9]. This paper presents an efficient and simple method for computing higher-order derivatives of the hypergeometric function. The remainder of this section covers the derivation of the recurrence formula. In Sec. 2 we discuss a two-level operator overloading technique based on the approach in the Diamant library and compare run times. Section 3 covers the application of our approach to an ionization problem. Conclusions are given in Sec. 4.

1.1 Low-Order Derivatives

For simplicity we rewrite (1) as

$$\alpha(z)\varphi^{(2)}(z)+\beta(z)\varphi^{(1)}(z)-\gamma\varphi(z)=0 \tag{4}$$

by setting $\alpha(z)=z(1-z)$, $\beta(z)=[c-(a+b+1)z]$, $\gamma=ab$, and from now on we assume $\alpha(z)\neq 0$. Furthermore we assume $z=z(x)$ to be n times differentiable with respect to x and write

$$v=v^{(0)}(x)=\varphi(z(x)) \tag{5}$$

Differentiating (5) with respect to x and omitting the dependence on x and z, we have

$$v^{(1)}=\varphi^{(1)}z^{(1)} \quad \text{and} \quad v^{(2)}=\varphi^{(2)}(z^{(1)})^2+\varphi^{(1)}z^{(2)}$$

Assuming $z^{(1)}\neq 0$, we obtain

$$\varphi^{(1)}=\frac{v^{(1)}}{z^{(1)}} \quad \text{and} \quad \varphi^{(2)}=\frac{v^{(2)}-\left(\frac{v^{(1)}}{z^{(1)}}\right)z^{(2)}}{(z^{(1)})^2}$$

which we substitute into (4) to write

$$\alpha\frac{v^{(2)}-(v^{(1)}/z^{(1)})z^{(2)}}{(z^{(1)})^2}+\beta\frac{v^{(1)}}{z^{(1)}}-\gamma v^{(0)}=0$$

Since we assumed $\alpha(z)\neq 0$, one deduces

$$\begin{aligned} v^{(0)} &= \varphi \\ v^{(1)} &= \varphi^{(1)}z^{(1)} \\ v^{(2)} &= \frac{\gamma v^{(0)}(z^{(1)})^3-\beta v^{(1)}(z^{(1)})^2+\alpha v^{(1)}z^{(2)}}{\alpha z^{(1)}} \end{aligned} \tag{6}$$

The derivatives of v can be computed by evaluating the ${}_2F_1$ function only twice rather than n times. Higher-order derivatives can be computed by overloading (6). This approach is discussed in Sec. 2.2. When $z^{(1)}=0$, differentiating (5) yields

$$\begin{aligned} v^{(0)} &= \varphi, \qquad v^{(1)}=0, \qquad v^{(2)}=\varphi^{(1)}z^{(2)} \\ v^{(3)} &= \varphi^{(3)}(z^{(1)})^3+3\varphi^{(2)}z^{(2)}z^{(1)}+\varphi^{(1)}z^{(3)}=\varphi^{(1)}z^{(3)} \\ v^{(4)} &= \cdots=3\varphi^{(2)}(z^{(2)})^2+\varphi^{(1)}z^{(4)} \end{aligned} \tag{7}$$

If $z^{(2)}\neq 0$, one can again find expressions for $\varphi^{(1)}$ and $\varphi^{(2)}$, substitute into (4), and write $v^{(4)}$ as

$$v^{(4)}=\frac{3\gamma v(z^{(2)})^3-3\beta v^{(2)}(z^{(2)})^2+\alpha v^{(2)}z^{(4)}}{\alpha z^{(2)}} \tag{8}$$

1.2 Higher-Order Formulas

General higher-order terms $v^{(n)}$ are derived from Faá di Bruno's formula, which can be expressed in terms of Bell polynomials $B_{n,k}(z^{(1)},\cdots,z^{(n-k+1)})$ (see also [2])

$$v^{(n)} = (\varphi \circ z)^{(n)} = \sum_{k=1}^{n} \varphi^{(k)} B_{n,k}(z^{(1)},\cdots,z^{(n-k+1)}) \tag{9}$$

where

$$B_{n,k}(z^{(1)},\cdots,z^{(n-k+1)}) = \sum \frac{n!}{j_1!\cdots j_{n-k+1}!}\left(\frac{z^{(1)}}{1!}\right)^{j_1}\cdots\left(\frac{z^{(n-k+1)}}{(n-k+1)!}\right)^{j_{n-k+1}}$$

and the sum is over all partitions of n into k nonnegative parts such that

$$j_1 + j_2 + \cdots j_{n-k+1} = k \quad \text{and} \quad j_1 + 2j_2 + \cdots (n-k+1)\, j_{n-k+1} = n \tag{10}$$

Further details may be found in [14]. One easily verifies that (9) correctly expresses the equations in (7).

Theorem 1. *Assuming $z^{(l)} = 0$ for $1 \le l < m$ and $z^{(m)} \neq 0$, one may simplify equation (9) written at order $2m$ as*

$$v^{(2m)} = \varphi^{(1)} z^{(2m)} + b_{2m}\varphi^{(2)}(z^{(m)})^2 \tag{11}$$

where $b_{2m} = \dfrac{(2m)!}{2!(m!)^2}$. Moreover, one may write

$$\varphi^{(2)} = \frac{v^{(2m)} - \varphi^{(1)} z^{(2m)}}{b_{2m}(z^{(m)})^2}$$

Proof. One notices that, for $k = 1$, the monomial $B_{2m,1}$

$$B_{2m,1}(z^{(1)},\cdots,z^{(2m)}) = \frac{(2m)!}{1!}\left(\frac{z^{(2m)}}{(2m)!}\right)^1 = z^{(2m)}$$

multiplied by $\varphi^{(1)}$ yields the first term of (11). For $k = 2$, the partition $j_l = 0$ $(\forall l \neq m)$, $j_m = 2$, is the only one that satisfies (10). One deduces the second term of (11):

$$\varphi^{(2)} B_{2m,2}(z^{(1)},\cdots,z^{(2m-1)}) = \varphi^{(2)}\frac{(2m)!}{2!}\left(\frac{z^{(m)}}{(m)!}\right)^2 = b_{2m}\varphi^{(2)}(z^{(m)})^2 \tag{12}$$

Other polynomials $B_{2m,k}(z^{(1)},\cdots,z^{(2m-k+1)})$ $(k > 2)$ vanish because they have at least one nonzero exponent j_l $(1 \le l < m)$ for which the respective basis $z^{(l)}$ was assumed to be 0. □

Theorem 2. *Assuming $z^{(l)} = 0$ for $1 \le l < m$ and $z^{(m)} \neq 0$, then the first $2m$ derivatives of the compound function v satisfy*

$$v^{(n)} = 0, \quad \forall n = 1,\cdots,m-1 \tag{13}$$

$$v^{(n)} = \varphi^{(1)} z^{(n)}, \quad \forall n = m,\cdots,2m-1 \tag{14}$$

$$v^{(2m)} = \frac{b_{2m}\gamma v (z^{(m)})^3 - b_{2m}\beta v^{(m)}(z^{(m)})^2 + \alpha v^{(m)} z^{(2m)}}{\alpha z^{(m)}} \tag{15}$$

Proof. We determine a recurrence over m for the first $2m-4$ derivatives of v and the Bell formulas as we did in Theorem 1 for the last ones. Following formula (14), we deduce

$$\varphi^{(1)} = v^{(m)}/z^{(m)} \tag{16}$$

The use of (16) and (12) in the ODE equation (4) leads to (15). □

2 Taylor Coefficient Propagation

Sections 1.1 and 1.2 show that the derivatives $v^{(n)}$ of the $v(x) = \varphi(z(x)) =_2 F_1(z(x))$ can be obtained from the first two derivatives $\varphi^{(1)}$ and $\varphi^{(2)}$. Now we need to provide an explicit formula for the Taylor coefficients that then can be implemented in an overloading library.

2.1 Series Computations

When the function of interest is the solution of an ODE, the usual approach starts with the Taylor series

$$v(x(t)) = v_0 + v_1 t + v_2 t^2 + v_3 t^3 + v_4 t^4 + \dots$$

and its derivatives

$$\begin{aligned} v^{(1)}(x) &= v_1 + 2v_2 t + 3v_3 t^2 + 4v_4 t^3 + \dots = \tilde{v}_1 + \tilde{v}_2 t + \tilde{v}_3 t^2 + \tilde{v}_4 t^3 + \dots \\ v^{(2)}(x) &= \tilde{v}_2 + 2\tilde{v}_3 t + 3\tilde{v}_4 t^2 + \dots \end{aligned}$$

for v. We can also write the respective series for $z, z^{(1)}, \dots$ and substitute them into the ODE, in our case into (4). Repeating the previous assumptions $z_1 \neq 0$ and $\alpha(z) = z(z-1) \neq 0$, we can apply this approach to the ${}_2F_1$ and eventually arrive[4] at

$$\begin{aligned} \sum_{i=0} (\mathbf{z}\tilde{\mathbf{v}}_2\tilde{\mathbf{z}}_1)_i t^i - \sum_{i=0} \Big(\sum_{j=0}^{i} z_j (\mathbf{z}\tilde{\mathbf{v}}_2\tilde{\mathbf{z}}_1)_{i-j} \Big) t^i = \\ ab \sum_{i=0} \Big(\sum_{j=0}^{i} v_j (\tilde{\mathbf{z}}_1^3)_{i-j} \Big) t^i - \\ c \sum_{i=0} (\tilde{\mathbf{v}}_1 \tilde{\mathbf{z}}_1^2)_i t^i + (a+b+1) \sum_{i=0} \Big(\sum_{j=0}^{i} z_j (\tilde{\mathbf{v}}_1 \tilde{\mathbf{z}}_1^2)_{i-j} \Big) t^i \\ + \sum_{i=0} (\mathbf{z}\tilde{\mathbf{v}}_1\tilde{\mathbf{z}}_2)_i t^i - \sum_{i=0} \Big(\sum_{j=0}^{i} z_j (\mathbf{z}\tilde{\mathbf{v}}_1\tilde{\mathbf{z}}_2)_{i-j} \Big) t^i \end{aligned} \tag{17}$$

We can now match coefficients for the t^i. For t^0 the match yields

$$\tilde{v}_2 = \frac{ab v_0 \tilde{z}_1^3 - c\tilde{v}_1 \tilde{z}_1^2 + (a+b+1) z_0 \tilde{v}_1 \tilde{z}_1^2 + z_0 \tilde{v}_1 \tilde{z}_2 - z_0^2 \tilde{v}_1 \tilde{z}_2}{z_0 \tilde{z}_1 - z_0^2 \tilde{z}_1} \tag{18}$$

The occurrence of $\tilde{v}_1$ in the right-hand side is the effect of the second-order equation. We need to compute that explicitly to start the recursion. As shown in Theorem 2, no more than two evaluations of the ${}_2F_1$ function are required. Taylor coefficients v_{i+2} ($i = 1, \dots$) are computed from the v_{i+1}. The main drawbacks are the complexity of the recursion obtained from (17) and the fact that, aside from convolutions on the Taylor coefficients, everything is specific to ${}_2F_1$ as our particular intrinsic of interest. In the following section we will look at a more generic alternative.

2.2 Overloading Strategies

For high-order differentiation tools, the relevant choices are AD02 for Fortran [15], Adol-C for C and C++ [10], and Rapsodia for both Fortran and C++ [7]. All of them rely on operator

[4] We left out some tedious intermediate steps; the $(\mathbf{p}_s \mathbf{q}_t)_r$ denote $\sum_{j=0}^{r} (p_{j+s} q_{i-j+t})$.

overloading as the vehicle of attaching derivative computations to the arithmetic operators and intrinsic functions provided by the programming language.

The ${}_2F_1$ function may be overloaded by using a Taylor coefficient recursion obtained from formula (18), intermediate computations such as $z_0\bar{v}_1\bar{z}_1^2$ being overloaded also. This two-level overloading using any of the three tools mentioned above can, depending on the application context, be inefficient. To illustrate the point, we consider the differentiation of the product $\phi : (x,y) \mapsto r = x * y$. According to the Leibniz rule this requires $(k+1)$ multiplications at order k. Because the general-purpose AD tools need to handle any sequence of intrinsics, they will often compute for each intrinsic *all* Taylor coefficients up to a user-defined but perhaps compile-time fixed order $K \geq k$. Consequently the number of multiplications for ϕ then rises to $(K+1)(K+2)/2$.

The nested convolutions in (18) imply at least an effort of $\mathscr{O}(K^3)$ for the computation of all the derivatives up to order K. Because of the complexity of (18) it is difficult to arrive at a more precise estimate, in particular when one considers the mix of intrinsic operations occurring in the recurrence and optimizing the computations by storing intermediate values.

A particular application scenario of interest here is the asymptotic numerical method (ANM), see [8, 6, 5], in which the derivative order rises with the ANM iteration count and at each iteration just the kth order derivatives have to be computed. The AD library *Diamant* (an acronym for **Di***fférentiation* **a***utomatique de la* **M***éthode* **a***symptotique* **n***umérique* **T***ypée*) is geared toward ANM with the goal of being as efficient as hand-coded ANM. It differs from the standard AD tools such as AD02 because it permits computing Taylor coefficients for just the particular desired order k at a time, not requiring the computation of lower orders $(l = 1,\cdots,k-1)$ or higher ones $(m = k+1,\cdots,K)$. Of course, for many intrinsics the computation of the kth-order Taylor coefficient of the result requires knowledge of all coefficients up to order k of the arguments. Rather than recomputing them, however, Diamant stores the highest coefficients at each ANM iteration so that they are available at the following iteration. Storing the coefficients up to order $k-1$ and subsequently computing order k is not unlike the approach of running Adol-C in forward mode once up to order $k-1$ while taping (that is, storing the coefficients) and following with the reverse mode for order k, which reads the stored coefficients. Adol-C, however, has no means to incrementally grow the order k on the tape in successive runs. As a consequence of the approach in Diamant the recurrence formulas for some intrinsics need to be modified. For instance, the typical recurrence for division $(w = u/v)$ makes use of the Taylor coefficients w_k of the result w. Such implementation details are discussed in the Diamant manual.

The Diamant approach can be applied to the efficient computation of Taylor coefficients of the hypergeometric function. In Table 1 we compare three AD strategies $\mathscr{D}_0$ – $\mathscr{D}_2$. All are implemented within Diamant. $\mathscr{D}_0$ implements a naive approach in which the recurrence formulas are evaluated by means of a loop over all orders k up to a statically fixed bound K. $\mathscr{D}_1$ performs the differentiation with a loop up to only the current order k of differentiation. Compared to $\mathscr{D}_0$ this avoids the computation of the unused high-order differentiations $(m = k+1,\cdots,K)$. Therefore $\mathscr{D}_1$ is comparable to the Adol-C implementation, although without the need for a tape.[5] A direct comparison with Adol-C was not possible, however, because Adol-C does not support the complex arithmetic used in ${}_2F_1$. The factor of improvement over $\mathscr{D}_0$ can be observed to be roughly three; see Table 1. $\mathscr{D}_2$ is the implementation specialized for the ANM. As discussed above, the differentiation is performed only at order k. Because we removed the loop over the order, we can expect a drop of a factor of K in the computational effort compared to the other versions. The run times give in Table 1 clearly show that the $\mathscr{D}_2$

[5] The tapeless version of Adol-C currently supports only first-order computations.

Table 1. Runtime comparison of the computation of ${}_2F_1$ derivatives at order k

k	$\mathscr{D}_0$	$\mathscr{D}_1$	$\mathscr{D}_2$
1	0.276	0.276	0.276
4	0.788	0.700	0.636
6	1.544	1.164	0.912
8	2.820	1.848	1.236
12	8.097	4.164	2.032
16	17.925	8.189	3.008
24	57.936	23.313	5.428
32	131.632	50.883	8.565

specialization for ANM yields substantial benefits. The results also agree with the expected complexity drop for $\mathscr{D}_2$.

3 The Ionization Application

The fully differential cross section (FDCS) on helium depends on the solid angles Ω_s, Ω_1, and Ω_2 for the scattered electron and the two ejected electrons and on the energies E_1 and E_2 of the ejected electrons. Assuming a unique interaction between the target and the incoming electron, one computes this FDCS as

$$\frac{\partial^5 \sigma}{\partial \Omega_s \partial \Omega_1 \partial \Omega_2 \partial E_1 \partial E_2} = \frac{k_1 k_2 k_s}{k_i} |M|^2 \tag{19}$$

where $\mathbf{k_i}$, $\mathbf{k_s}$, $\mathbf{k_1}$, and $\mathbf{k_2}$ denote, respectively, the momenta of incident, scattered, first ejected, and second ejected electrons. The matrix element M is a 9-dimensional integral

$$M = \frac{1}{2\pi} \int \psi_f^*(\mathbf{r_1}, \mathbf{r_2}) e^{i\mathbf{k_s}.\mathbf{r_0}} V \psi_i(\mathbf{r_1}, \mathbf{r_2}) e^{i\mathbf{k_i}.\mathbf{r_0}} d\mathbf{r_0} d\mathbf{r_1} d\mathbf{r_2} \tag{20}$$

where $V = -2r_0^{-1} + |\mathbf{r_0} - \mathbf{r_1}|^{-1} + |\mathbf{r_0} - \mathbf{r_2}|$ is the Coulomb interaction between the projectile and the helium atom, r_0 is the distance between the incident electron and the nucleus, and r_1 and r_2 are the distances between one of the helium electrons and its nucleus. The wavefunctions ψ_i and ψ_f are the solutions of the Schrödinger equation for the helium atom. No exact formulas exist for ψ_i and ψ_f. The well-known Bethe transformation, $e^{i\mathbf{k}.\mathbf{a}} k^{-2} = 4\pi^{-1} \int e^{i\mathbf{k}.\mathbf{r}} |\mathbf{r} - \mathbf{a}|^{-1} d\mathbf{r}$, allows for the integration on $\mathbf{r_0}$. Thus, the computation of (20) needs a six-dimensional integral only.

On the one hand, the bound state wavefunction ψ_i may be approximated, under the first Born approximation, by means of a Hylleraas-type wavefunction. As an example, let us consider the *GRN* wavefunction involving an increasing number N of parameters

$$\varphi_{GRN}(r_1, r_2, r_{12}) = \sum_{i,j,k \geq 0} c_{ijk} (r_1^i r_2^j r_{12}^k + r_1^j r_2^i r_{12}^k) r_{12}^k \tag{21}$$

where coefficients c_{ijk} were determined from the solution of a minimization problem. Nonzero coefficients are indicated in Table 2. On the other hand, the best approximation for the final state ψ_f is that of [3], which satisfies exact asymptotic boundary collisions. The two numerical approaches available to tackle an accurate Hylleraas wavefunction are either the use of three

Table 2. Nonzero c_{ijk} and their respective order k appearing in GRN for various N

	N				
c_{ijk}	3	5	9	14	k
c_{000}	×	×	×	×	3
c_{200}	×	×	×	×	5
c_{220}		×	×	×	7
c_{300}		×	×	×	3
c_{320}			×	×	8
c_{400}	×			×	7
c_{002}		×	×	×	5
c_{202}			×	×	7
c_{222}				×	9
c_{302}			×	×	8
c_{402}			×	×	9
c_{003}				×	6
c_{223}				×	10
c_{004}				×	7

${}_1F_1$ functions and a six-dimensional numerical quadrature [12] (expensive in computer time) or one occurrence of the ${}_2F_1$ function and a two-dimensional quadrature applied to high-order derivative tensors [3].

The gain in number of integrals has to be paid. The Brauner's method is based on a term $D = e^{-ar_1}e^{-br_2}e^{-cr_{12}}/(r_1r_2r_{12})$ whose third-order derivative yields the simple wavefunction $\psi_i(\mathbf{r_1},\mathbf{r_2}) = e^{-ar_1}e^{-br_2}e^{-cr_{12}}$. This enables the writing of terms $r_1^i r_2^j r_{12}^k e^{-ar_1}e^{-br_2}e^{-cr_{12}}$ appearing when using Brauner's method as derivatives of D.

3.1 Implementations

For this comparison we use four different implementations of the differentiation of the original code `P` with the ${}_2F_1$ function. These are compared on the computation of the derivatives appearing in the GRN functions.

`P_O`: This implementation enables the computation of any mixed derivative of order less than or equal to six. The differentiation has been fully hand coded for some of the statements, whereas a few Fortran functions (multiplication, division, and the hypergeometric function) replicate operator overloading AD. The derivative computation is organized, as in Diamant, by increasing order of Taylor coefficients; that is, the differentiation is done first at order 1, then at order 2, and so on. Consequently, the classical recurrence formulas were split into six derivative functions (one per order) to avoid useless recalculations. The ${}_2F_1$ derivatives are implemented Faá di Bruno's formula. Most of this code is, however, proprietary and used here only for verification and as a basis for run-time comparison.

$\texttt{P}_{\mathscr{R}}$: The original function evaluation code `P` is overloaded by means of the Rapsodia library. Rapsodia supports computations with complex numbers in a Fortran program and provides higher-order derivatives in the standard AD fashion, that is, by propagating all Taylor coefficients up to a given order. The ${}_2F_1$ function is overloaded by using Faá di Bruno's formula up to order 8. Thus $\texttt{P}_{\mathscr{R}}$ enables, by means of linear combinations [4] or high-order tensors [11], the computation of any derivative even beyond the desired order of 10.

$\mathtt{P}_{\mathscr{D}_0}$: The original function evaluation code `P` is overloaded by using Rapsodia, while the ${}_2F_1$ function is overloaded by using $\mathscr{D}_0$,

$\mathtt{P}_{\mathscr{D}_2}$: The original function evaluation code `P` is overloaded by using Rapsodia, while the ${}_2F_1$ function is overloaded by using $\mathscr{D}_2$.

Table 3 shows the run times for GRN with $N = 14$. Because the derivatives in `P_O` are hand-coded only up to order six, it cannot compute all coefficients. Nevertheless, we use `P_O` to perform a fourth-order differentiation with respect to r_{12}, and we overload it using Rapsodia or Diamant to obtain sixth-order derivatives in r_1 and r_2 by a linear combination of six directions. This permits the computation of the 10^{order} derivative related to coefficient c_{223}. For comparison we also show the run times of `P` when the ${}_2F_1$ is commented out to indicate the complexity of computing it.

Table 3. CPU time consumptions for the GRN function

N	`P` without ${}_2F_1$	`P_O`	$\mathtt{P}_{\mathscr{R}}$	$\mathtt{P}_{\mathscr{D}_2}$	$\mathtt{P}_{\mathscr{D}_0}$
14	2.812	×	5.9	4.7	8.4

Numerical results presented in Table 3 prove the efficiency of the Diamant approach. More precisely, the two-level overloading strategy together with the $\mathscr{D}_2$ library is about 25% less time consuming than the Faá di Bruno formula implementation realized in $\mathtt{P}_{\mathscr{R}}$. Physical results we obtained are in good agreement with an "absolute" experiment [13]. They will be published in another paper.

While the inspection of the physical results indicates the principal correctness of our implementation, one might still wonder about the effects of the approximation to ${}_2F_1$ itself upon which all of the above is built. It would be difficult to compute the actual error, but we can, for instance, look at the discrepancy between the Taylor coefficients of the original generic Brauner term from the application and the Taylor coefficients of the manually coded first derivative. The relative discrepancies are shown in Fig. 1. We observe the pronounced effect

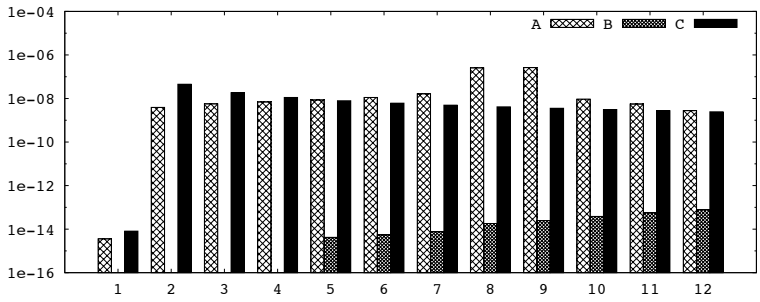

Fig. 1. Maximum relative discrepancy in either the real or the imaginary part of Taylor coefficients for the Brauner generic term (case A), for the Brauner generic term without ${}_2F_1$ (case B), and just the ${}_2F_1$ function itself (case C) computed for orders $k \in [1, 12]$

of the computations involving the ${}_2F_1$ approximation while the code without it has an exact floating-point match even up to order four.

4 Conclusions

We investigated the different options for computing derivatives of the hypergeometric function ${}_2F_1(a,b,c;z)$ and the consequences of vanishing Taylor coefficients of the argument z. The run-time comparison shows the advantage of the Taylor coefficient recurrence over Faá di Bruno's formula. We showed various options for a generic two-level operator overloading strategy. For the ${}_2F_1$ function we show how the latter can be used together with either the equations (13)–(15) or the Taylor coefficients from formula (18) matched for t^0. Furthermore we demonstrated the benefits of the specialization of the AD approach for ANM implemented in the Diamant library for ${}_2F_1$ and we also looked at the run time comparison of the computation of derivatives for an ionization application that involves the ${}_2F_1$ function.

Acknowledgement. Jean Utke was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy under Contract DE-AC02-06CH11357.

References

1. Abramowitz, M., Stegun, I.: Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables. Dover, New York (1972)
2. Bell, E.: Exponential polynomials. Ann. of Math. **35**, 258–277 (1934)
3. Brauner, M., Briggs, J., Klar, H.: Triply-differential cross sections for ionization of hydrogen atoms by electrons and positrons. J. Phys. B.: At. Mol. Phys. **22**, 2265–2287 (1989)
4. Charpentier, I., Cappello, C.D.: High order cross derivative computation for the differential cross section of double ionization of helium by electron impact. Tech. Rep. 5546, INRIA (2005)
5. Charpentier, I., Lejeune, A.: The Diamant library for an efficient automatic differentation of the asymptotic numerical method. AD08 (submitted)
6. Charpentier, I., Potier-Ferry, M.: Différentiation automatique de la Méthode asymptotique numérique Typée: l'approche Diamant. Comptes Rendus Mécanique (accepted)
7. Charpentier, I., Utke, J.: Fast higher order derivative tensors. Preprint, Argonne National Laboratory (2007). Under review at OMS; also available as preprint ANL/MCS-P1463-1107
8. Cochelin, B., Damil, N., Potier-Ferry, M.: Méthode Asymptotique Numérique. Hermes Science Publications (2007)
9. Griewank, A.: Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation. No. 19 in Frontiers in Applied Mathematics. SIAM, Philadelphia (2000)
10. Griewank, A., Juedes, D., Utke, J.: ADOL–C, a package for the automatic differentiation of algorithms written in C/C++. ACM Trans. Math. Software **22**(2), 131–167 (1996)
11. Griewank, A., Utke, J., Walther, A.: Evaluating higher derivative tensors by forward propagation of univariate Taylor series. Mathematics of Computation **69**, 1117–1130 (2000)
12. Jones, S., Madison, D.: Single and double ionization of atoms by photons, electrons, and ions. In: AIP Conference proceedings 697, pp. 70–73 (2003)
13. Lahmam-Bennani, A., Taouil, I., Duguet, A., Lecas, M., Avaldi, L., Berakdar, J.: Origin of dips and peaks in the absolute fully resolved cross sections for the electron-impact double ionizaton of helium. Phys. Rev. A. **59**(5), 3548–3555 (1999)

14. Noschese, S., Ricci, P.: Differentiation of multivariable composite functions and Bell polynomials. J. Comput. Anal. Appl. **5**(3), 333–340 (2003)
15. Pryce, J., Reid, J.: AD01, a Fortran 90 code for automatic differentiation. Tech. Rep. RAL-TR-1998-057, Rutherford Appleton Laboratory, Chilton, Oxfordshire, England (1998)